TIP 352: Tcl Style Guide

core.tcl-lang.org/tips/doc/trunk/tip/352.md

Abstract

This document describes a set of conventions that it is suggested people use when writing Tcl code. It is substantially based on the Tcl/Tk Engineering Manual [247].

NOTE

A transcription of the original version (dated August 22, 1997) of this file into PDF is available online at http://www.tcl.tk/doc/styleGuide.pdf - Donal K. Fellows.

Introduction

This is a manual for people who are developing Tcl code for Wish or any other Tcl application. It describes a set of conventions for writing code and the associated test scripts. There are three reasons for the conventions. First, the conventions ensure that certain important things get done; for example, every procedure must have documentation that describes each of its arguments and its result, and there must exist test scripts that exercise every line of code. Second, the conventions guarantee that all of the Tcl and Tk code has a uniform style. This makes it easier for us to use, read, and maintain each other's code. Third, the conventions help to avoid some common mistakes by prohibiting error-prone constructs such as building lists by hand instead of using the list building procedures.

This document is based heavily on the *Tcl/Tk Engineering Manual* written by John Ousterhout. John's engineering manual specified the style of the C code used in the implementation of Tcl/Tk and many of its extensions. The manual is very valuable to the development of Tcl/Tk and is an important reason why Tcl is a relatively easy system to maintain.

Deciding any style standard involves making trade-offs that are usually subjective. This standard was created in an iterative process involving the Tcl/Tk group at Sun Laboratories. I don't claim that these conventions are the best possible ones, but the exact conventions don't really make that much difference. The most important thing is that we all do things the same way.

Please write your code so that it conforms to the conventions from the very start. For example, don't write comment-free code on the assumption that you'll go back and put the comments in later once the code is working. This simply won't happen. Regardless of how good your intentions are, when it comes time to go back and put in the comments you'll find that you have a dozen more important things to do; as the body of

uncommented code builds up, it will be harder and harder to work up the energy to go back and fix it all. One of the fundamental rules of software is that its structure only gets worse over time; if you don't build it right to begin with, it will never get that way later.

The rest of this document consists of 8 major parts. We start with Section 2 which discusses executable files. Section 3 discusses the overall structure of packages and namespaces. Section 4 describes the structure of a Tcl code file and how to write procedure headers. Section 5 desribes the Tcl naming conventions. Section 6 presents low-level coding conventions, such as how to indent and where to put curly braces. Section 7 contains a collection of rules and suggestions for writing comments. Section 8 describes how to write and maintain test suites. Section 9 contains a few miscellaneous topics, such as keeping a change log.

Executable files

An executable is a file, collection of files, or some other collection of Tcl code and necessary runtime environment. Often referred to as applications, an executable is simply what you run to start your program. The format and exact make up of an executable is platform-specific. At some point, however, a Tcl *start-up script* will be evaluated. It is the start-up script that will bootstrap any Tcl based application.

The role of the start-up script is to load any needed *packages*, set up any non-package specific state, and finally start the Tcl application by calling routines inside a Tcl package. If the start-up script is more than a few lines it should probably be a package itself.

There are several ways to create executable scripts. Each major platform usually has a unique way of creating an executable application. Here is a brief description of how these applications should be created on each platform:

1. The most common method for creating executable applications on UNIX platforms is the infamous **#!** mechanism built into most shells. Unfortunately, the most common approach of just giving a path to wish is not recommended. Don't do:

```
#! /usr/local/tclsh8.0 -f "$0" "$@"
```

> This method will not work if the file **tclsh** is another script that, for example, locates and starts the most recent version of Tcl. It also requires **tclsh** to be in a particular place, which makes the script less portable. Instead, the following method should be used which calls **/bin/sh** which will in turn exec the **wish** application.

```
#!/bin/sh
# the next line restarts using wish \
exec wish8.0 "$0" "$@"
```

> This example will actually locate the **wish** application in the user's path which can be very useful for developers. The backslash is recognized as part of a comment to *sh*, but in Tcl the backslash continues the comment into the next line which keeps the *exec* command from executing again. However, more stable sites would probably want to include the full path instead of just **wish**. Note that the version number of the **tclsh** or **wish** interpreter is usually added to the end of the program name. This allows you use a specific version of Tcl. In addition, many sites include a link of **wish** to the latest version currently installed. This is useful if you know that your code will work on any version of Tcl.

2. On the Windows platform you only need to end a file with the **.tcl** extension and the file will be run when the user double clicks on the file. This is, of course, assuming you have installed Tcl/Tk.

> Alternatively, you may create a **.bat** file which explicitly executes **tclsh** or **wish** with an absolute path to your start-up script. Please check the Windows documentation for more details about **.bat** files.

3. The Macintosh platform doesn't really have a notion of an executable Tcl file. One of the reasons for this is that, unlike UNIX or Windows, you can only run one instance of an application at a time. So instead of callingwish with a specific script to load, we must create a copy of the **wish** application that is tied to our script.

> The easiest way to do this is to use the application *Drag&Drop Tclets* or the *SpecTcl* GUI builder which can do this work for you. You can also do this by hand by putting the start-up script into a TEXT resource and name it *tclshrc* - which ensures it gets sourced on start-up. This can be done with *ResEdit* (a tool provided by Apple) or other tools that manipulate resources. Additional scripts can also be placed in TEXT resource to make the application completely contained.

Packages and namespaces

Tcl applications consist of collections of *packages*. Each package provides code to implement a related set of features. For example, Tcl itself is a package, as is Tk; these packages happen to be implemented in both C and Tcl. Other packages are implemented completely in Tcl such as the **http** package included in the Tcl distribution. Packages are the units in which code is developed and distributed: a single package is typically developed by a single person or group and distributed as a unit. It is possible to combine many independently-developed packages into a single application; packages should be designed with this in mind. The notion of *namespaces* were created to help make this easier. Namespaces help to hide private aspects of packages and avoid name collisions. A package will generally export one public namespace which will include all state and routines that are associated with the package. A package should not contain any global variables or global procedures. Side effects when loading a package should be avoided. This document will focus on packages written entirely in Tcl. For a discussion of packages built in C or C and Tcl see the *Tcl/Tk Engineering Manual*.

Package names

Each package should have a unique *name*. The name of the package is used to identify the package. It is also used as the name of the namespace that the package exports. It is best to have a simple one word name in all lower-case like **http**. Multi-word names are ok as well. Additional words should just be concatenated with the first word but start with a capital letter like **specMenu**.

Coming up with a unique name for your package requires a collaborative component. For internal projects this is an easy task and can usually be decided among the management or principal engineers in your organization. For packages you wish to publish, however, you should make an effort to make sure that an existing package isn't already using the same name you are. This can often be done by checking the comp.lang.tcl newsgroup or the standard Tcl ftp sites. It is also suggested (but not required) that you register your name on the NIST Identifier Collaboration Service (NICS). It is located at: http://pitch.nist.gov/nics

Version numbers

Each package has a two-part version number such as 7.4. The first number (7) is called the major version number and the second (4) is called the minor version number. The version number changes with each public release of the package. If a new release contains only bug fixes, new features, and other upwardly compatible changes, so that code and scripts that worked with the old version will also work with the new version, then the minor version number increments and the major version number stays the same (e.g., from 7.4 to 7.5). If the new release contains substantial incompatibilities, so that existing code and scripts will have to be modified to run with the new version, then the major version number increments and the minor version number resets to zero (e.g., from 7.4 to 8.0).

Package namespaces

As of version 8.0, Tcl supports namespaces to hide the internal structure of a package. This helps avoid name collisions and provides a simpler way to manage packages. All packages written for Tcl 8.0 or newer should use namespaces. The name of the name space should be the same as the package name.

Structure

There are a couple of ways to deploy a package of Tcl commands.

- A **pkgIndex.tcl** file is used to create *packages* that can be loaded on demand by any Tcl script. Like a **tclIndex** file, a package specifies a set of Tcl and/or shared libraries that can be loaded when needed. A package, however, must be explicitly requested by using the **package require** command. You can use the **pkg_mkIndex** command to create a package index file for your use. In most cases, particularly in code you distribute to others, it is better to use a package instead of the **tclIndex** auto-loading mechanism.
- On the Macintosh platform, shared libraries can be made into self contained packages. You simply need to add a TEXT resource with the name of **pkgIndex**. It will be treated in the exact same fashion as a **pkgIndex.tcl** file. The **pkgIndex** resource should have the same format as the **pkgIndex.tcl** file.

How to organize a code file

Each source code file should either contain an entire application or a set of related procedures that make up a package or a another type of identifiable module, such as the implementation of the menus for your application, or a set of procedures to implement HTTP access. Before writing any code you should think carefully about what functions are to be provided and divide them into files in a logical way. The most manageable size for files is usually in the range of 500-2000 lines. If a file gets much larger than this, it will be hard to remember everything that the file does. If a file is much shorter than this, then you may end up with too many files in a directory, which is also hard to manage.

The file header

The first part of a code file is referred to as the *header*. It contains overall information that is relevant throughout the file. It consists of everything but the definitions of the file's procedures. The header typically has four parts, as shown below:

```
           /   # specMenu.tcl --
           |   #
Abstract   |   #       This file implements the Tcl code for creating and
           |   #       managing the menus in the SpecTcl application.
           \   #
           /   # Copyright (c) 1994-1997 Sun Microsystems, Inc.
           |   #
Copyright  |   # See the file "license.terms" for information on usage and
           |   # redistribution of this file, and for a DISCLAIMER OF ALL
           \   # WARRANTIES.
               #
Revision       # SCCS: %Z% %M% %I% %E% %U%
String         # RCS: Id

           /   package require specTable
Package    |   package provide specMenu 1.0
Definition |   namespace eval specMenu {
           |       namespace export addMenu
           |       array set menuData {one two three}
           |       ...
           \   }
```

Abstract: The first few lines give the name of the file and a brief description of the overall functions provided by the file, just as in header files.

Copyright notice: The notice protects ownership of the file. The copyright shown above is included in the Tcl and Tk sources. More product specific packages would probably have the words *All rights reserved included* instead. If more than one entity contributed to the page they should each have a distinct copyright line.

Revision string: The contents of this string are managed automatically by the source code control system for the file, such as RCS or SCCS (both are shown in the example). It identifies the file's current revision, date of last modification, and so on.

Package definition: Also any **require** statements for other packages that this package depends on should be the first code in the file. Any global variables that are managed by this file should be declared at the top of the page. The name space definition should be next and the export list should be the first item in the namespace definition.

Please structure your header pages in exactly the order given above and follow the syntax of the example as closely as possible. The file **fileHead.tcl** [*not available*] provides a template for a header page.

Multi-file packages

Some packages may be too large to fit into one file. You may want to consider breaking the package into multiple independent packages. However, when that is not an option you need to make one of the files the *primary* file. The primary file will include the complete export list and the definitions of all exported variables and procedures. The secondary files should only contain supporting routines to the primary file. It is important to construct

your package in this manner or utilities like **pkg_mkIndex** will not work correctly. Finally, the header to the various files should make it clear which file is the primary file and which are supporting files.

Procedure headers

After the header you will have one or more procedures. Each procedure will begin with a *procedure header* that gives overall documentation for the procedure, followed by the declaration and body for the procedure. See below for an example.

```
# tcl::HistRedo --
#
#       Fetch the previous or specified event, execute it, and then
#       replace the current history item with that event.
#
# Arguments:
#       event   (optional) index of history item to redo.  Defaults
#                to -1, which means the previous event.
# Results:
#        The result is that of the command being redone.  Also replaces
#        the current history list item with the one being redone.
proc tcl::HistRedo {{event -1}} {
    ...
}
```

The header should contain everything that a caller of the procedure needs to know in order to use the procedure, and nothing else. It consists of three parts:

Abstract: The first lines in the header give the procedure's name, followed by a brief description of what the procedure does. This should not be a detailed description of how the procedure is implemented, but rather a high-level summary of its overall function. In some cases, such as callback procedures, I recommend also describing the conditions under which the procedure is invoked and who calls the procedure.

Arguments: This portion of the header describes the arguments that the procedure expects. Each argument should get at least one line. The comment should describe the expected type and describe it's function. Optional arguments should be pointed out and the default behavior of an unspecified argument should be mentioned. Comments for all of the arguments should line up on the same tab stop.

Results: The last part of the header describes the value returned by the procedure. The type and the intended use of the result should be described. This section should also mention any *side effects* that are worth noting.

The file **tclProcHead** [*not available*] contains a template for a procedure header which should be used as a base for all new Tcl commands. Follow the syntax of the above example exactly (same indentation, double-dash after the procedure name, etc.).

Procedure declarations

The procedure declaration should also follow exactly the syntax in the example above. Note that the procedure is defined outside the namespace command that defines the export list and namespace globals. The first line gives the **proc** keyword, the procedure name, and an argument list. If there are many arguments, they may spill onto additional lines (see Sections 6.1 and 6.3 for information about indentation).

Parameter order

Procedure parameters may be divided into three categories. *In* parameters only pass information into the procedure (either directly or by pointing to information that the procedure reads). *Out* parameters point to things in the caller's memory that the procedure modifies such as the name of a variable the procedure will modify. *In-out* parameters do both. Below is a set of rules for deciding on the order of parameters to a procedure:

1. Parameters should normally appear in the order in, in/out, out, except where overridden by the rules below.

2. If an argument is actually a sub-command for the command than it should be the first argument of the command. For example:

```
proc graph::tree {subCmd args} {
    switch $subCmd {
        add {
            eval add_node $args
        }
        draw {...
```

3. If there is a group of procedures, all of which operate on an argument of a particular type, such as a file path or widget path, the argument should be the first argument to each of the procedures (or after the sub-command argument).

Procedure bodies

The body of a procedure follows the declaration. See Section 6 for the coding conventions that govern procedure bodies. The curly braces enclosing the body should be on different lines, as shown in the examples above, even if the body of the procedure is empty.

Naming conventions

Choosing names is one of the most important aspects of programming. Good names clarify the function of a program and reduce the need for other documentation. Poor names result in ambiguity, confusion, and error. This section gives some general principles to follow when choosing names and lists specific rules for name syntax, such as capitalization.

General considerations

The ideal variable name is one that instantly conveys as much information as possible about the purpose of the variable it refers to. When choosing names, play devil's advocate with yourself to see if there are ways that a name might be misinterpreted or confused. Here are some things to consider:

1. Are you consistent? Use the same name to refer to the same thing everywhere. For example, within the code for handling standard bindings in Tk widgets, a standard name **w** is always used to refer to the window associated with the current event.

2. If someone sees the name out of context, will they realize what it stands for, or could they confuse it with something else? For example, the procedure name **buildStructure** could get confused with some other part of the system. A name like **buildGraphNode** both describes what part of the system it belongs to and what it is probably used for.

3. Could this name be confused with some other name? For example, it's probably a mistake to have two variables **str** and **string** in the same procedure: it will be hard for anyone to remember which is which. Instead, change the names to reflect their functions. For example, if the strings are used as source and destination for a copy operation, name them **src** and **dst**.

4. Is the name so generic that it doesn't convey any information? The variable **str** from the previous paragraph is an example of this; changing its name to **src** makes the name less generic and hence conveys more information.

Basic syntax rules

Below are some specific rules governing the syntax of names. Please follow the rules exactly, since they make it possible to determine certain properties of a variable just from its name.

1. Exported names for both procedures and variables always start with a *lower*-case letter. Procedures and variables that are meant only for use with in the current package or namespace should start with an *upper*-case letter. We chose lower-case for the exported symbols because it is possible they may be commonly used from the command line and they should be easy to write. For example:

```
# CountNum is a private variable
set CountNum 0
# The function addWindow is public
proc addWindow {} {...
# newWindow is a public interface in the spectcl namespace
proc spectcl::newWindow {} {...
```

2. In multi-word names, the first letter of each trailing word is capitalized. Do not use underscores or dashes as separators between the words of a name.

```
set numWindows 0
```

3. Any variable whose value refers to another variable has a name that ends in **Name**. Furthermore, the name should also indicate what type of variable the name is referring to. These names are often used in arguments to procedures that are taking a name of a variable.

```
proc foo::Bar {arrayName} {
    upvar 1 $arrayName array
    ...
}
```

4. Variables that hold Tcl code that will be **eval**ed should have names ending in **Script**.

```
proc log::eval {logScript} {
    if {$Log::logOn} {
        set result [catch {eval $logScript} msg]
        ...
```

5. Variables that hold a partial Tcl command that must have additional arguments appended before being a valid script should have names ending in **Cmd**.

```
foreach scrollCmd $listScrollCmds {
    eval $scrollCmd $args
}
```

Low-level coding conventions

This section describes several low-level syntactic rules for writing Tcl code. These rules help to ensure that all of the Tcl code looks the same, and they prohibit a few confusing coding constructs.

Indents are 4 spaces

Each level of indentation should be four spaces. There are ways to set 4-space indents in all of the most common editors. Be sure that your editor really uses four spaces for the indent, rather than just displaying tabs as four spaces wide; if you use the latter approach then the indents will appear eight spaces wide in other editors.

Code comments occupy full lines

Comments that document code should occupy full lines, rather than being tacked onto the ends of lines containing code. The reason for this is that side-by-side comments are hard to see, particularly if neighboring statements are long enough to overlap the side-by-side comments. Also it is easy to place comments in a place that could cause errors. Comments must have exactly the structure shown in the example below, with a blank line above and below the comment. The leading blank line can be omitted if the comment is at the beginning of a block, as is the case in the second comment in the example below. Each comment should be indented to the same level as the surrounding code. Use proper English in comments: write complete sentences, capitalize the first word of each sentence, and so on.

```
# If we are running on the Macintosh platform then we can
# assume that the sources are located in the resource fork
# of our application, and we do not need to search for them.
# Note that there is a blank line below it to separate it
# more strongly from the code.

if {$tcl_platform(platform) == "macintosh"} {
    return
}

foreach dir $dirList {
    # If the source succeds then we are done.
    # Note there is no blank line above the comment;
    # the indentation change is visible enough.

    if {![catch {source [file join $dir file.tcl]}]} {
        break
    }
}
```

Continuation lines are indented 8 spaces

You should use continuation lines to make sure that no single line exceeds 80 characters in length. Continuation lines should be indented 8 spaces so that they won't be confused with an immediately-following nested block. Pick clean places to break your lines for continuation, so that the continuation doesn't obscure the structure of the statement. For example, if a procedure call requires continuation lines, try to avoid situations where a single argument spans multiple lines. If the test for an **if** or **while** command spans lines, try to make each line have the same nesting level of parentheses and/or brackets if possible. I try to start each continuation line with an operator such as *, **&&**, or **||**; this makes it clear that the line is a continuation, since a new statement would never start with such an operator.

Only one command per line

You should only have one Tcl command per line on the page. Do not use the semi-colon character to place multiple commands on the same line. This makes the code easier to read and helps with debugging.

Curly braces: { goes at the end of a line

Open curly braces can not appear on lines by themselves in Tcl. Instead, they should be placed at the end of the preceding line. Close curly braces are indented to the same level as the outer code, i.e., four spaces less than the statements they enclose. However, you shouldalways use curly braces rather than some other list generating mechanism that will work in the Tcl language. This will help make code more readable, will avoid unwanted side effects, and in many cases will generate faster code with the Tcl compiler.

Control structures should always use curly braces, even if there is only one statement in the block. Thus you shouldn't write code like

```
if {$tcl_platform(platform) == "unix"} return
```

but rather

```
if {$tcl_platform(platform) == "unix"} {
    return
}
```

This approach makes code less dense, but it avoids potential mistakes like unwanted Tcl substitutions. It also makes it easier to set breakpoints in a debugger, since it guarantees that each statement is on a separate line and can be named individually.

Parenthesize expressions

Use parentheses around each subexpression in an expression to make it absolutely clear what is the evaluation order of the expression (a reader of your code should not need to remember Tcl's precedence rules). For example, don't type

```
if {$x > 22 && $y <= 47} ...
```

Instead, type this:

```
if {($x > 22) && ($y <= 47)} ...
```

Always use the return statement

You should always explicitly use the **return** statement to return values from a Tcl procedure. By default Tcl will return the value of the last Tcl statement executed in a Tcl procedure as the return value of the procedure which often leads to confusion as to where the result is coming from. In addition, you should use a **return** statement with no argument for procedures whose results are ignored. Supplying this return will actually speed up your application with the new Tcl compiler. For example, don't write code like this:

```
proc foo {x y} {
    if {$x < 0} {
        incr x
    } else {
        expr $x + $y
    }
}
```

But rather, type this:

```
proc foo {x y} {
    if {$x < 0} {
        return [incr x]
    } else {
        return [expr $x + $y]
    }
}
```

For Tcl procedures that have no return value a single **return** statement with no arguments is placed at the end of the procedure.

Switch statements

The **switch** statement should be formatted as below. Always use the **--** option to avoid having the string be confused with an option. This can happen when the string is user generated. Comments can be added on the same line as the pattern to comment the pattern case. The comments for each case should line up on the same tab stop and must be within the braces. Note that this is an exception to the standard commenting conventions.

```
switch -regexp -- $string {
    plus -
    add {       # Do add task
        ...
    }
    subtract {  # Do subtract case
        ...
    }
    default {
        ...
    }
}
```

If statements

Never use the **then** word of an **if** statement. It is syntactic sugar that really isn't that useful. However, the **else** word should always be used as it does impart some semantic information and it is more like the C language. Here is an example:

```
if {$x < 0} {
    ...
} elseif {$x == 0} {
    ...
} else {
    ...
}
```

Documenting code

The purpose of documentation is to save time and reduce errors. Documentation is typically used for two purposes. First, people will read the documentation to find out how to use your code. For example, they will read procedure headers to learn how to call the procedures. Ideally, people should have to learn as little as possible about your code in order to use it correctly. Second, people will read the documentation to find out how your code works internally, so they can fix bugs or add new features; again, good documentation will allow them to make their fixes or enhancements while learning the minimum possible about your code. More documentation isn't necessarily better: wading through pages of documentation may not be any easier than deciphering the code. Try to pick out the most important things that will help people to understand your code and focus on these in your documentation.

Document things with wide impact

The most important things to document are those that affect many different pieces of a program. Thus it is essential that every procedure interface, every structure declaration, and every global variable be documented clearly. If you haven't documented one of these things it will be necessary to look at all the uses of the thing to figure out how it's supposed to work; this will be time-consuming and error-prone.

On the other hand, things with only local impact may not need much documentation. For example, in short procedures I don't usually have comments explaining the local variables. If the overall function of the procedure has been explained, and if there isn't much code in the procedure, and if the variables have meaningful names, then it will be easy to figure out how they are used. On the other hand, for long procedures with many variables I usually document the key variables. Similarly, when I write short procedures I don't usually have any comments in the procedure's code: the procedure header provides enough information to figure out what is going on. For long procedures I place a comment block before each major piece of the procedure to clarify the overall flow through the procedure.

Don't just repeat what's in the code

The most common mistake I see in documentation (besides it not being there at all) is that it repeats what is already obvious from the code, such as this trivial (but exasperatingly common) example:

```
# Increment i.

incr i
```

Documentation should provide higher-level information about the overall function of the code, helping readers to understand what a complex collection of statements really means. For example, the comment

```
# Probe into the array to see if the symbol exists.
```

is likely to be much more helpful than

```
# Loop through every array index, get the third value of the
# list in the content to determine if it has the symbol we are
# looking for. Set the result to the symbol if we find it.
```

Everything in this second comment is probably obvious from the code that follows it.

Another thing to consider in your comments is word choice. Use different words in the comments than the words that appear in variable or procedure names. For example, the comment

```
# SwapPanels --
#
# Swap the panels.
# ...
```

is not a very useful comment. Everything in the comment is already obvious from the procedure's name. Here is a much more useful comment:

```
# SwapPanels --
#
# Unmap the current UI panel from the parent frame and replace
# it with the newly specified frame. Make sure that the new
# panel fits into the old frame and resize if needed.
# ...
```

This comment tells *why* you might want to use the procedure, in addition to *what* it does, which makes the comment much more useful.

Document each thing in exactly one place

Systems evolve over time. If something is documented in several places, it will be hard to keep the documentation up to date as the system changes. Instead, try to document each major design decision in exactly one place, as near as possible to the code that implements the design decision. The principal documentation for each procedure goes in the procedure header. There's no need to repeat this information again in the body of the procedure (but you might have additional comments in the procedure body to fill in details not described in the procedure header). If a library procedure is documented thoroughly in a manual entry, then I may make the header for the procedure very terse, simply referring to the manual entry.

The other side of this coin is that every major design decision needs to be documented *at least* once. If a design decision is used in many places, it may be hard to pick a central place to document it. Try to find a data structure or key procedure where you can place the main body of comments; then reference this body in the other places where the decision is used. If all else fails, add a block of comments to the header page of one of the files implementing the decision.

Write clean code

The best way to produce a well-documented system is to write clean and simple code. This way there won't be much to document. If code is clean, it means that there are a few simple ideas that explain its operation; all you have to do is to document those key ideas. When writing code, ask yourself if there is a simple concept behind the code. If not, perhaps you should rethink the code. If it takes a lot of documentation to explain a piece of code, it is a sign that you haven't found a clean solution to the problem.

Document as you go

It is extremely important to write the documentation as you write the code. It's very tempting to put off the documentation until the end; after all, the code will change, so why waste time writing documentation now when you'll have to change it later? The problem is that the end never comes - there is always more code to write. Also, the more undocumented code that you accumulate, the harder it is to work up the energy to document it. So, you just write more undocumented code. I've seen many people start a project fully intending to go back at the end and write all the documentation, but I've never seen anyone actually do it.

If you do the documentation as you go, it won't add much to your coding time and you won't have to worry about doing it later. Also, the best time to document code is when the key ideas are fresh in your mind, which is when you're first writing the code. When I write new code, I write all of the header comments for a group of procedures before I fill in any of the bodies of the procedures. This way I can think about the overall structure and how the pieces fit together before getting bogged down in the details of individual procedures.

Document tricky situations

If code is non-obvious, meaning that its structure and correctness depend on information that won't be obvious to someone reading it for the first time, be sure to document the non-obvious information. One good indicator of a tricky situation is a bug. If you discover a subtle property of your program while fixing a bug, be sure to add a comment explaining the problem and its solution. Of course, it's even better if you can fix the bug in a way that eliminates the subtle behavior, but this isn't always possible.

Testing

One of the environments where Tcl works best is for testing. While Tcl has traditionally been used for testing C code it is equally as good at testing other Tcl code. Whenever you write new code you should write Tcl test scripts to go with that code and save the tests in files so that they can be re-run later. Writing test scripts isn't as tedious as it may sound. If you're developing your code carefully you're already doing a lot of testing; all you need to do is type your test cases into a script file where they can be reused, rather than typing them interactively where they vanish after they're run.

Basics

Tests should be organized into script files, where each file contains a collection of related tests. Individual tests should be based on the procedure **test**, just like in the Tcl and Tk test suites. Here are two examples:

```
test expr-3.1 {floating-point operators} {
    expr 2.3*.6
} 1.38
test expr-3.2 {floating-point operators} {unixOnly} {
    list [catch {expr 2.3/0} msg] $msg
} {1 {divide by zero}}
```

test is a procedure defined in a script file named **defs**, which is **source**d by each test file. **test** takes four or five arguments: a test identifier, a string describing the test, an optional argument describing the conditions under which this test should run, a test script, and the expected result of the script. **test** evaluates the script and checks to be sure that it produces the expected result. If not, it prints a message like the following:

```
==== expr-3.1 floating-point operators
==== Contents of test case:
    expr 2.3*.6
==== Result was:
1.39
---- Result should have been:
1.38
---- expr-3.1 FAILED
```

To run a set of tests, you start up the application and **source** a test file. If all goes well no messages appear; if errors are detected, a message is printed for each error.

The test identifier, such as **expr-3.1**, is printed when errors occur. It can be used to search a test script to locate the source for a failed test. The first part of the identifier, such as **expr**, should be the same as the name of the test file, except that the test file should have a **.test** extension, such as **expr.test**. The two numbers allow you to divide your tests into groups. The tests in a particular group (e.g., all the **expr-3.***n* tests) relate to a single sub-feature, such as a single procedure. The tests should appear in the test file in the same order as their numbers.

The test name, such as **floating-point operators**, is printed when errors occur. It provides human-readable information about the general nature of the test.

Before writing tests I suggest that you look over some of the test files for Tcl and Tk to see how they are structured. You may also want to look at the **README** files in the Tcl and Tk test directories to learn about additional features that provide more verbose output or restrict the set of tests that are run.

Organizing tests

Organize your tests to match the code being tested. The best way to do this is to have one test file for each source code file, with the name of the test file derived from the name of the source file in an obvious way (e.g. **http.test** contains tests for the code in **http.tcl**).

Within the test file, have one group of tests for each procedure (for example, all the **http-3.*n*** tests in **http.test** are for the procedure **http::geturl**). The order of the tests within a group should be the same as the order of the code within the procedure. This approach makes it easy to find the tests for a particular piece of code and add new tests as the code changes.

The Tcl test suite was written a long time ago and uses a different style where there is one file for each Tcl command or group of related commands, and the tests are grouped within the file by sub-command or features. In this approach the relationship between tests and particular pieces of code is much less obvious, so it is harder to maintain the tests as the code evolves. I don't recommend using this approach for new tests.

Coverage

When writing tests, you should attempt to exercise every line of source code at least once. There will be occasionally be code that you can't exercise, such as code that exits the application, but situations like this are rare. You may find it hard to exercise some pieces of code because existing Tcl commands don't provide fine enough control to generate all the possible execution paths. In situations like this, write one or more new Tcl commands just for testing purposes. It's much better to test a facility directly then to rely on some side effect for testing that may change over time. Use a similar approach in your own code, where you have an extra file with additional commands for testing.

It's not sufficient just to make sure each line of code is executed by your tests. In addition, your tests must discriminate between code that executes correctly and code that isn't correct. For example, write tests to make sure that the **then** and **else** branches of each **if** statement are taken under the correct conditions. For a loop, run different tests to make the loop execute zero times, one time, and two or more times. If a piece of code removes an element from a list, try cases where the element to be removed is the first element, last element, only element, and neither first element nor last. Try to find all the places where different pieces of code interact in unusual ways, and exercise the different possible interactions.

Fixing bugs

Whenever you find a bug in your code it means that the test suite wasn't complete. As part of fixing the bug, you should add new tests that detect the presence of the bug. I recommend writing the tests after you've located the bug but *before* you fix it. That way you can verify that the bug happens before you implement the fix and the bug doesn't happen afterwards, so you'll know you've really fixed something. Use bugs to refine your testing approach: think about what you might be able to do differently when you write tests in the future to keep bugs like this one from going undetected.

Tricky features

I also use tests as a way of illustrating the need for tricky code. If a piece of code has an unusual structure, and particularly if the code is hard to explain, I try to write additional tests that will fail if the code is implemented in the obvious manner instead of using the tricky approach. This way, if someone comes along later, doesn't understand the documentation for the code, decides the complex structure is unnecessary, and changes the code back to the simple (but incorrect) form, the test will fail and the person will be able to use the test to understand why the code needs to be the way it is. Illustrative tests are not a substitute for good documentation, but they provide a useful addition.

Test independence

Try to make tests independent of each other, so that each test can be understood in isolation. For example, one test shouldn't depend on commands executed in a previous test. This is important because the test suite allows tests to be run selectively: if the tests depend on each other, then false errors will be reported when someone runs a few of the tests without the others.

For convenience, you may execute a few statements in the test file to set up a test configuration and then run several tests based on that configuration. If you do this, put the setup code outside the calls to thetest procedure so it will always run even if the individual tests aren't run. I suggest keeping a very simple structure consisting of setup followed by a group of tests. Don't perform some setup, run a few tests, modify the setup slightly, run a few more tests, modify the setup again, and so on. If you do this, it will be hard for people to figure out what the setup is at any given point and when they add tests later they are likely to break the setup.

Miscellaneous

Porting issues

Writing portable scripts in Tcl is actually quite easy as Tcl itself is quite portable. However, issues do arise that may require writing platform specific code. To conditionalize your code in this manner you should use the **tcl_platform** array to determine platform specific differences. You should avoid the use of theenv variable unless you have already determined the platform you are running on via the **tcl_platform** array.

As Tcl/Tk has become more cross platform we have added commands that aid in making your code more portable. The most common porting mistakes result from assumptions about file names and locations. To avoid such mistakes always use the **file join** command and list commands so that you will handle different file separation characters or spaces in file names. In Tk, you should always use provided high level dialog boxes instead or creating your own. The **font** and **menu** commands has also be revamped to make writing cross-platform code easier.

Changes files

Each package should contain a file namedchanges that keeps a log of all significant changes made to the package. The **changes** file provides a way for users to find out what's new in each new release, what bugs have been fixed, and what compatibility problems might be intro- duced by the new release. The **changes** file should be in chronological order. Just add short blurbs to it each time you make a change. Here is a sample from the Tk **changes** file:

```
5/19/94 (bug fix) Canvases didn't generate proper Postscript for
stippled text. (RJ)

5/20/94 (new feature) Added "bell" command to ring the display's
bell. (JO)

5/26/94 (feature removed) Removed support for "fill" justify mode
from Tk_GetJustify and from the TK_CONFIG_JUSTIFY configuration
option.  None of the built-in widgets ever supported this mode
anyway. (SS)
*** POTENTIAL INCOMPATIBILITY ***
```

The entries in the **changes** file can be relatively terse; once someone finds a change that is relevant, they can always go to the manual entries or code to find out more about it. Be sure to highlight changes that cause compatibility problems, so people can scan the **changes** file quickly to locate the incompatibilities. Also be sure to add your initials to the entry so that people scanning the log will know who made a particular change.

(The Tcl and Tk core additionally uses a ChangeLog file that has a much higher detail within it. This has the advantage of having more tooling support, but tends to be so verbose that the shorter summaries in the changes file are still written up by the core maintainers before each release.)